UNE IMP EHE CWI ★QMIPS★ INR UER ZAR UTO

Quantitative Modeling In Parallel Systems

Failure detection algorithms for a reliable execution of parallel programs

Sophie Chabridon
EHEI, Université René Descartes
45 rue des Saints-Pères
75270 PARIS CEDEX 06, France

Erol Gelenbe
Department of Electrical Engineering
Duke University
Durham, N.C. 27708-0291

D W3.T2.18-4.v1

19 September 1995

Distribution Level: All partners

Approved by: E. Gelenbe

Abstract We discuss a novel technique for improving the dependability of parallel programs executing on an MIMD shared memory architecture. The idea is to empower certain tasks of each application program to carry out failure detection, and to reschedule the execution of those tasks which are considered to have failed. The source of failures is assumed to be any event which prevents a processor from performing the task that was assigned to it. This can be the actual stoppage of the processor due to some failure, or it may be the

The QMIPS Project: The QMIPS project is a collaborative research project supported by the CEC as ESPRIT-BRA project no 7269. It is being carried out by the following organisations: CWI (Amsterdam), EHEI (University of Paris V), Imperial College (London), INRIA (Sophia-Antipolis), University of Erlangen, University of Newcastle, University of Torino and University of Zaragoza.

preemption of the processor by a high priority task which is not part of the application program considered and which impedes the progress of a task of the application program. The technique we propose is based on a task graph representation of the parallel program, in which communications between tasks have been voluntarily isolated to the end of each task which is being considered. We represent the parallel application programs being considered by a task graph which takes into account the precedence constraints between tasks. We propose and evaluate several algorithms which can detect failures and restart failed tasks based on knowledge of this graph structure. A discrete-event simulator is used to evaluate the performance under the effect of failures, with the use of our detection and restart algorithms, of a specific parallel application: the Fast Fourier Transform. We measure the overhead due to the detection algorithm and the total execution time – including failure detection and recovery – obtained for different processor failure rates.

1 Introduction

Parallel and distributed systems, because of their size and complexity, are more prone than sequential machines to incidents which impair program execution. In addition to conventional failures which can occur in such systems, specific events may occur which can cause a particular application program's execution to fail. For instance:

- Certain processing units may be pre-empted unexpectedly by other programs,

- Certain processing units may be turned off by users who are unaware of the ongoing application,

- Workload surges from higher priority applications may slow down a particular application's work on certain processing units, creating excessive delay for some tasks, which can then be interpreted as a failure. In some cases, such surges are known to eliminate certain processes from memory.

In sequential processing, dependability is of paramount importance only in restricted classes of applications. For many critical applications, special purpose techniques have been suggested; among them we can cite log-based recovery for databases [1] and recovery block mechanisms for sequential programs [18].

In the case of distributed systems, these techniques have been enriched to provide distributed robustness to failures. Roll-back recovery based techniques have been suggested [16, 2, 6, 17, 20, 21] and also distributed recovery blocks [15].

In recent research [11, 12, 13, 14] new techniques have been developed which guarantee that a parallel program of the fork-join or threads and barriers type can be modified so that processor failures can be detected, and processing rescheduled when processor failures occur. This is achieved by a *diagnosis* (or detection) algorithm (which incurs some overhead) with a *rescheduling* algorithm aimed at redistribution of task responsibilities from the unreliable processors to their reliable counterparts. Both of these algorithms act *dynamically* either concurrently with the execution of the application program, or alternating with it. These existing theoretical results have not been evaluated in terms of actual performance. However asymptotic algorithmic analysis indicate that they can be implemented at acceptable cost.

The approach we propose is inspired from the above work. It differs from the checkpointing and roll-back recovery techniques, and also from distributed algorithm techniques, which are essentially system oriented (rather than application oriented). It does not rely on system level constructs for failure recovery – such system oriented constructs would in general exist at a "lower" level. Furthermore it does not rely on a specific model of parallel computation, but can be implemented in any parallel or distributed program, as long as the program can be decomposed into a set of communicating tasks. Specific MIMD architectures where our approach applies includes the KSR (Kendall Square Research) machine, the CM-5, as well as distributed networks of workstations running programs under systems such as PVM.

The idea we pursue is that of empowering all – or a subset – of tasks in the program with failure detection capabilities. Those tasks which can carry out failure detection – and which we call *agents* – do so for other tasks. If an agent determines that some task has not completed, then it will also test that task's predecessors, until a task is found which has failed but whose predecessors have not failed. The agent then restarts the failed task on a running (i.e. non-failed) PU; after that it goes on with its work of failure detection.

1.1 Effects of failures when there is no failure detection

The desirability of failure detection and recovery in computer systems is usually well admitted. However quantitative measures of the effect of failures when recovery algorithms are either not available, or are slow to act, can provide insight into the importance of such techniques. Therefore in this section we will present simulation results to illustrate the need for failure detection and recovery algorithms in distributed programs. We consider parallel programs running on a MIMD system which is subject to failures. The program is composed of a set of interdependent tasks, and when a PU on which a task is running fails, then the corresponding task is stopped. When a PU fails it remains unavailable for some time F, after which it becomes operational again (it is "repaired"). In these simulations, we consider two alternatives concerning task restart after the PU is again operational (i.e. recovers) after a failure:

- If there is no automatic restart of tasks, then - since we do not have failure detection and subsequent recovery - the program as a whole will have failed when any processor fails.
- In the case with automatic restart, the task which was running on the failed PU is automatically restarted when the PU becomes operational again. Consequently, even without failure detection, tasks are restarted and the program as a whole has a chance to complete. The program's time to completion will depend on the failure rate and on the down time F.

Automatic restart could exist in a distributed system where some form of system level checkpointing restarts the failed PU, and then this is followed by the execution of a routine which restarts all tasks which failed when the PU failed.

On Figure 1 we plot results from simulations we have carried out; we will not discuss the simulation model used in detail, since our purpose here is merely to motivate the subsequent work. These results deal with the percentage of programs which completed successfully, against the failure rate γ, for 1000 simulated programs selected at random. In order to normalize the time units involved, each individual task of a program has an exponentially distributed execution time of average $\tau = 1$. The failure rate γ ranges from 0.0001 to 0.1 in logarithmic scale. Each simulated program contains $K = 50$ tasks. In order to concentrate the analysis on the reliability issues, these simulations assume that the number of PUs is unlimited and that communication times between tasks are small enough to be neglected. A program is stopped if it does not complete after a time $T_{max} = 200$. The results shown concern the case where there is no restart, and with automatic restart for four values of F: $10, 50, 100, 200$.

In the case with restart, the percentage of completed programs decreases significantly when $F = 200$; for $\gamma = 0.1$, less than 10 % of the programs complete. When tasks are not restarted automatically, the percentage of completed programs decreases quickly as the

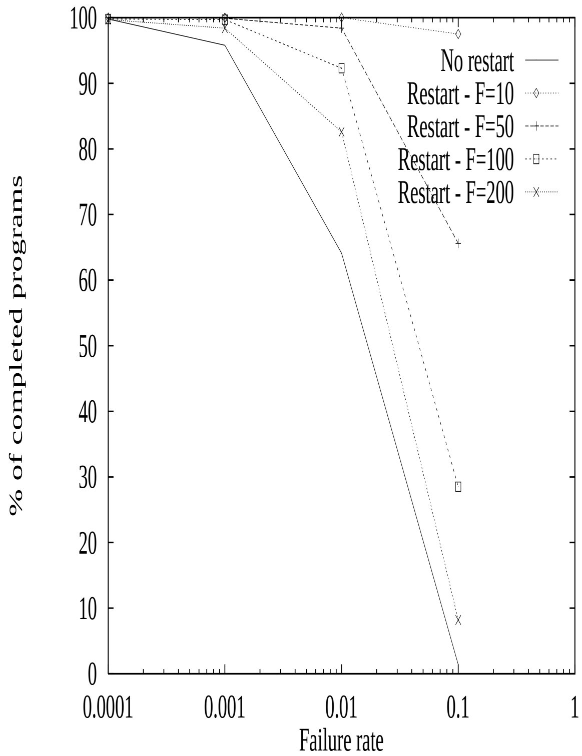


Figure 1. Effects of failures without detection

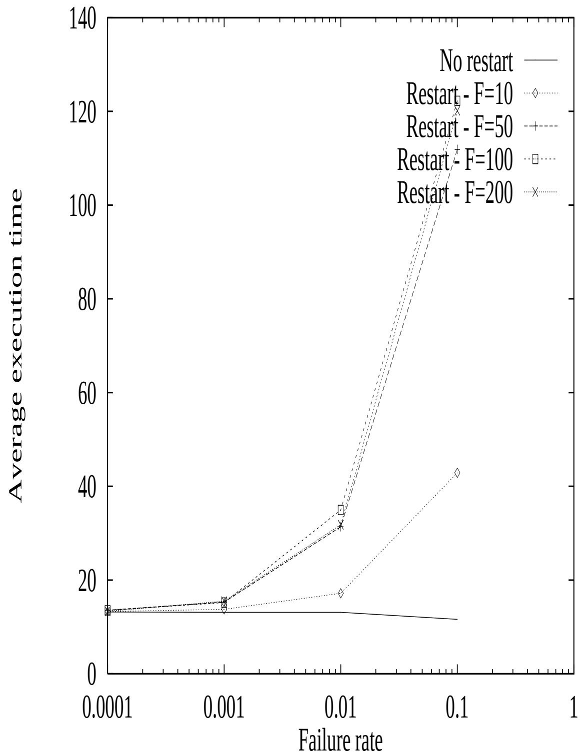


Figure 2. Average execution time of completed programs

failure rate increases, and almost no program completed when $\gamma = 0.1$. Figure 2 shows the average execution time of the *successfully completed programs*, in the presence of failures and without the benefit of detection algorithms. Without restart, programs which complete in the presence of failures are simply those for which no PUs fail during their execution. Shorter programs are more likely to avoid failures: average execution time of completed programs decreases from 13.17 when $\gamma = 0.0001$ to 11.6 when $\gamma = 0.1$. With automatic restart the programs' average execution time increases rapidly as a function of the failure rate, attaining more than 10 times their original value when $\gamma = 0.1$. These simulations strongly motivate the need for better methods for handling failures, which will detect failures and reschedule tasks rapidly.

2 Detection and recovery algorithms

A distributed application is composed of many interdependent tasks, and each task is a granule of sequential code which is executed on a single PU. Tasks are related to each other by precedence constraints, by the exchange of data, or of messages, and by the transfer of control. A task can be in one of four states. A task with no predecessor, a source task, is set to the state READY. A task having predecessors is initially in the state BLOCKED, and it passes to the state READY as soon as all its predecessors are completed. When a processor becomes available, a ready task becomes ACTIVE. After it has completed normally its execution its state is set to FINISHED. Whenever a task successfully completes, it must inform its successors in the task graph of this. Throughout this study, we will assume that either communications take place through a small failsafe shared memory or via perfectly reliable communication network.

We propose to select a subset of the tasks in the program, and make them responsible for executing the monitoring – i.e. failure detection and task reallocation – algorithm. These selected tasks will complete the computation they would carry out normally, and then actively detect failures, and assist the rest of the program's recovery from failures. We will call these selected tasks *agents*. Clearly, an agent can only examine its sibling tasks for failure, i.e. those for which it is neither a predecessor nor a successor. The schemes we propose will guarantee full dependability only if at least one of the processor is operational. In particular we will consider the following algorithms, varying essentially in the way agents are selected.

Algorithm 1 – Here only tasks which have no successor (i.e. the *leaves* of the task graph) are agents. Each leaf which has finished execution will first examine the other leaves of the task graph. For each leaf seen not to be in the BLOCKED state, the agent will *investigate* it by examining its predecessors. If a task has been in the ACTIVE state for *too long*, it is supposed to have failed. It will be removed from its current processor and restarted on another one. This permits to take into account real processor failures where the processor has simply stopped running and also cases where the processor is overloaded and execution progress becomes too slow. The choice of the time-out parameter determining the duration a task should not exceed is very important. A very small value may cause false alarms while a too large value would delay failure detection for too long.

We have considered in previous work [3, 4] the case of *fail-stop* processors as proposed in [19] where a processor stopping execution informs the rest of the system of this. The instant of failure is thus perfectly known by the other running tasks. This is no longer the case in this paper.

Algorithm 2 – Here we choose a certain number of intermediate tasks also to be agents, in addition to the leaves. The purpose is to detect failures soon enough to avoid that the distributed application fails completely before the leaves are reached. In most cases, any failure will be detected by the intermediate tasks without having to wait for leaves to be activated. Let the *rank* of a task be the longest path from itself back to a source task (a source being a task without predecessors). We may select some ranks, and assign the role of agent to some tasks in each rank. An agent will examine other tasks having its own rank, and then those of lesser rank, in order to detect failures. We call this look-back detection.

Algorithm 3 – This is an extension of Algorithm 2. Here, agents will carry out look-ahead detection for failures, in addition to the look-back of Algorithm 2. When an agent finds a task of the same rank which has completed its execution, it will test its successors. The purpose is to detect failures as soon as possible and to favor the rapid and successful completion of the whole program.

The assumptions concerning the manner in which failures may occur, as well as their nature (intermittent or permanent) are important. We consider the two relevant failure assumptions:

Failure Assumption A (FAA) – Tasks cannot autonomously restart. Any task whose processor *has not failed* by the end of its computational step will not subsequently fail until all of the program terminates. Thus any task which carries out the role of agent is preserved from failures. This is clearly a *best case* assumption, and would be valid if the agents' detection and restart functions were carried out on highly reliable special PUs.

Failure Assumption B (FAB) – Tasks cannot autonomously restart. Agents which have terminated their normal work, and which are now carrying out failure detection and restart for other tasks, can fail as well. This assumption is more realistic.

3 Simulations

We simulate the following model of a set of parallel programs running on a multiprocessor architecture. Each program is represented by a *Task-graph*; this model has been successfully used to represent parallel programs and to study their performance [7, 8, 9, 10]. In the task-graph model, a program is represented by a set of K tasks numbered $\{1, \ldots, \mathrm{K}\}$; each task is an indivisible unit of code and data which has to be executed sequentially. An acyclic graph G with K nodes represents the precedence relations between the tasks. The graph is acyclic because we are dealing with a computation which must eventually terminate. Each task i has an execution time t_i on a PU, and it may be executed on any one of the PUs in the same amount of time. When a task finishes its execution it informs all of its successors in the precedence graph, either by sending a message or through a shared memory. A task can begin execution when it is informed that all of its predecessors have terminated their execution.

In the simulations, each task graph representing a program, is generated at random. A probability p will be selected, and used to determine whether some task j, with $j > i$ is a successor of task i. By not allowing a task j to be a successor of some task $k < j$, we will guarantee that the graph is acyclic. By choosing a small value of p, we will simulate programs which are very "parallel" since any task will have few successors, while a larger value of p will limit the amount of parallelism in the program. In the simulation of random graphs we used $p = 0.1$ and $K = 50$ for the size of the graphs. For the sake of simplicity, the failure process is assumed to be Poisson, and all the durations are exponentially distributed unless otherwise stated.

The simulations have to account for the time spent by an agent both in detection, and in failure recovery: each time a task is tested by an agent, a constant time w is added to the execution time of the agent. When a task is restarted after a failure is detected, it takes an additional constant time C for the agent to do so.

3.1 Overhead of the detection algorithms without failures

Let us now evaluate and compare the failure detection Algorithms 1, 2 and 3. We evaluate the overhead due to each detection algorithm when there is no failure. Figure 3 presents simulation measurements for the average execution time of 10 different programs as a function of the program size in number of tasks K. The detection and restart durations are respectively $w = 0.1$ and $C = 1$. For algorithms 2 and 3, we show the results for $\pi = 0.5$, which means that half of the tasks are agents on the average, and for $\pi = 1$. In this latter case, all the tasks are agents and are generating overhead by running a detection algorithm; this is the worst case giving an upper bound of the overhead that can be generated by a detection algorithm when there is no failure.

Without failure, Algorithm 1 is the one which has the smallest overhead. Algorithm 3 has more overhead than Algorithm 2. For $K = 200$, the average execution time obtained with algorithm 3 and $\pi = 1$ is 180 when it is only 40 without detection. Thus, Algorithm 3 increases the average execution time by 140, i.e. more than five times the cost of 28 of Algorithm 2 with $\pi = 1$.

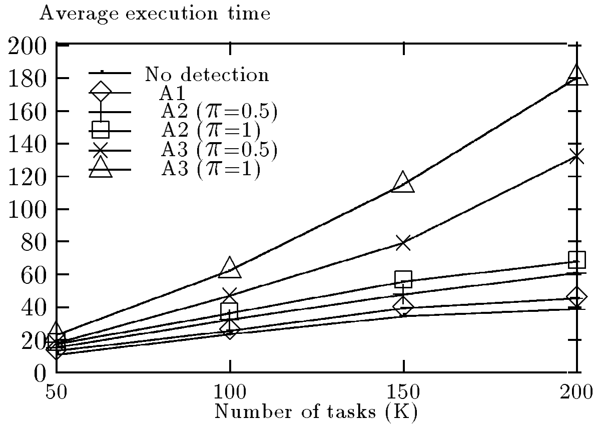


Figure 3. Cost of the detection algorithms without failures.

3.2 Comparison of the detection algorithms

We now consider the case where there are failures and the detection algorithms are used. In our simulation model, failures follow a Poisson process of rate γ ranging in $[0.0001, 0.1]$. Task execution times are exponentially distributed of average $\tau = 1$. Figure 4 presents average simulation results over 10 random graphs of 50 tasks for both failure assumptions A and B. We have used three values for the time-out on the task duration. On Figure 4a, the time-out is taken to be equal to τ. This means that each time a task is tested by an agent it will be restarted since it is not supposed to be completed before a duration of τ. This is a worst case illustrating the effect of false alarms. On Figure 4b, the time-out is taken to be $5 * \tau$

and on Figure 4c, it is $10 * \tau$.

For Algorithms 2 and 3, in addition to the leaves, tasks are selected to be agents using a probabilistic assignment. Therefore we have run simulations with four values of the probability π for selecting a rank of tasks to be agents: 0.25, 0.5, 0.75 and 1. On Figure 4, we only show one point for a particular algorithm at each failure rate value. It corresponds to the lowest execution time respectively to π. Under FAA, algorithm 2 gives the best result with $\pi = 0.25$ for $\gamma = 0.0001$ and $\gamma = 0.001$, and with $\pi = 1$ for $\gamma = 0.01$ and $\gamma = 0.1$. Still under FAA, algorithm 3 is plotted with $\pi = 0.25$ for $\gamma = 0.0001, 0.001$ and 0.01 and with $\pi = 1$ for $\gamma = 0.1$.

For failure assumption FAB, algorithm 2 gives the best results with $\pi = 0.25$ for $\gamma = 0.0001$ and $\gamma = 0.001$, with $\pi = 1$ for $\gamma = 0.01$ and with $\pi = 0.75$ for $\gamma = 0.1$; algorithm 3 is plotted with $\pi = 0.25$ for each value of γ.

Under failure assumption A, Algorithm 1 gives the lowest program execution time for each value of γ with a time-out which 10 times the average execution time τ. For instance, when γ is equal to 0.1, the program execution time is less than 200 units of time. With the same algorithm, it can reach 680 for a time-out of 1 unit of time. This shows that when tasks are restarted prematurely, the overall program execution time is more than doubled.

Under failure assumption B, the best results are obtained with a time-out of 10 times τ, as for failure assumption A. Algorithm 2 gives the lowest execution time for γ in the range $[0.001, 0.01]$. However when γ is equal to 0.1, algorithm 3 is better and gives an execution time of almost 310 units of time.

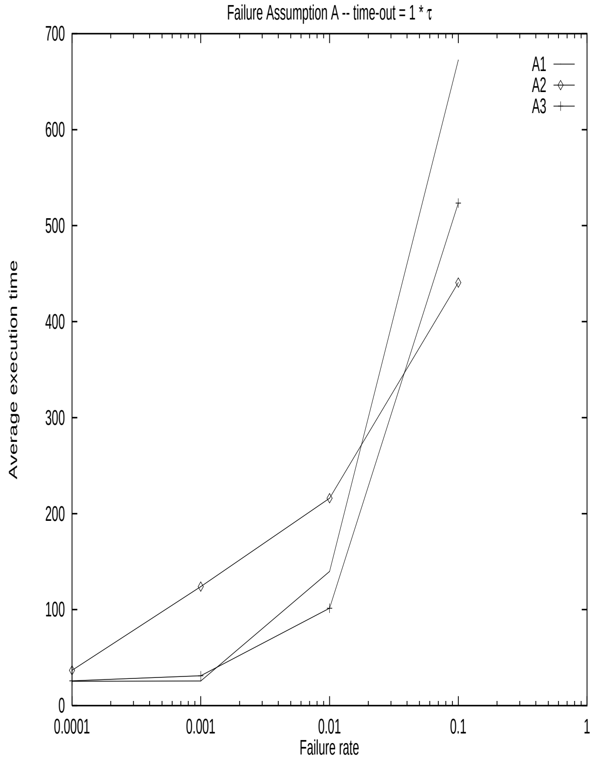
4 An example: dependable execution of the parallel FFT algorithm

In order to proceed further in the understanding of how our proposed failure detection and recovery algorithms work, and the manner in which they affect a practical parallel application, we will consider the discrete Fourier transform which is widely used in a variety of application areas, ranging from statistics to signal processing. Much effort has been spent by researchers to find efficient implementations of this algorithm, especially on parallel machines. In particular, the Fast Fourier Transform algorithm [5] is an efficient version which we consider in parallel form in this section.

We first describe the FFT algorithm, its parallelization as well as the corresponding task graph structure. Then we conduct simulations of the execution of the FFT task graph on a set of failing parallel processors with our failure detection and recovery algorithms.

For a given complex vector X, its discrete Fourier transform is given on a finite number of input samples by:

$$y(k) = \sum_{i=0}^{N-1} x(i)\omega_N^{ik}, \qquad k = 0, 1, ..., N-1$$



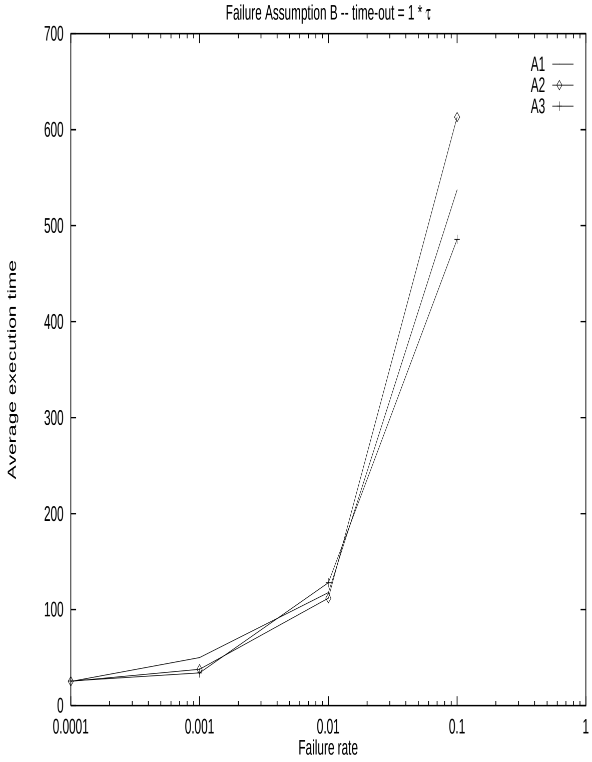
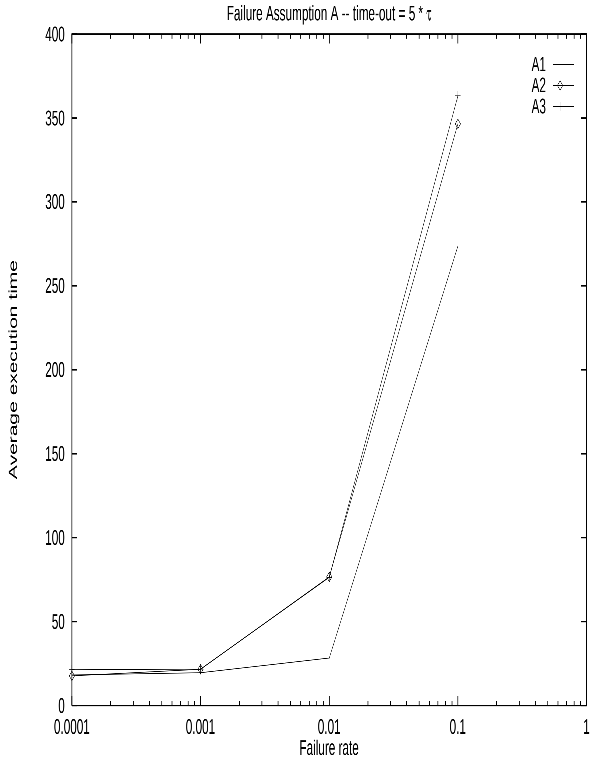


Figure 4a. Time-out of τ



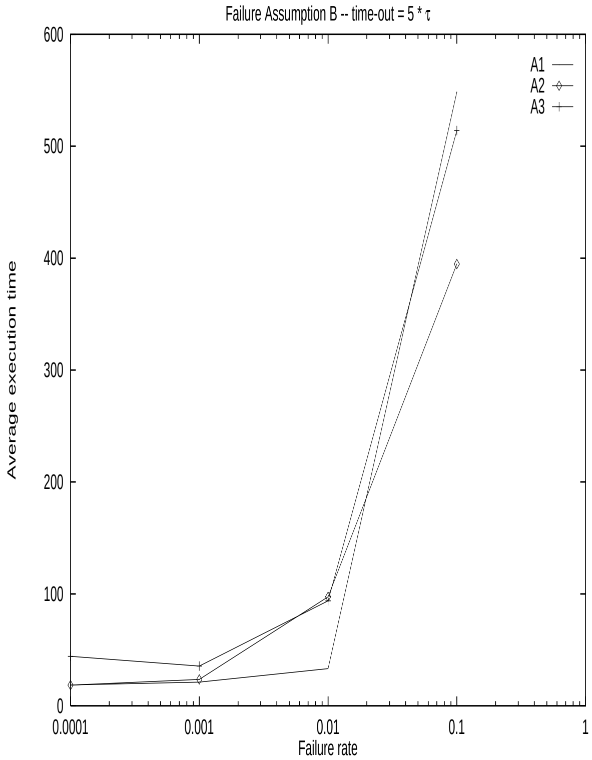


Figure 4b. Time-out of $5 * \tau$

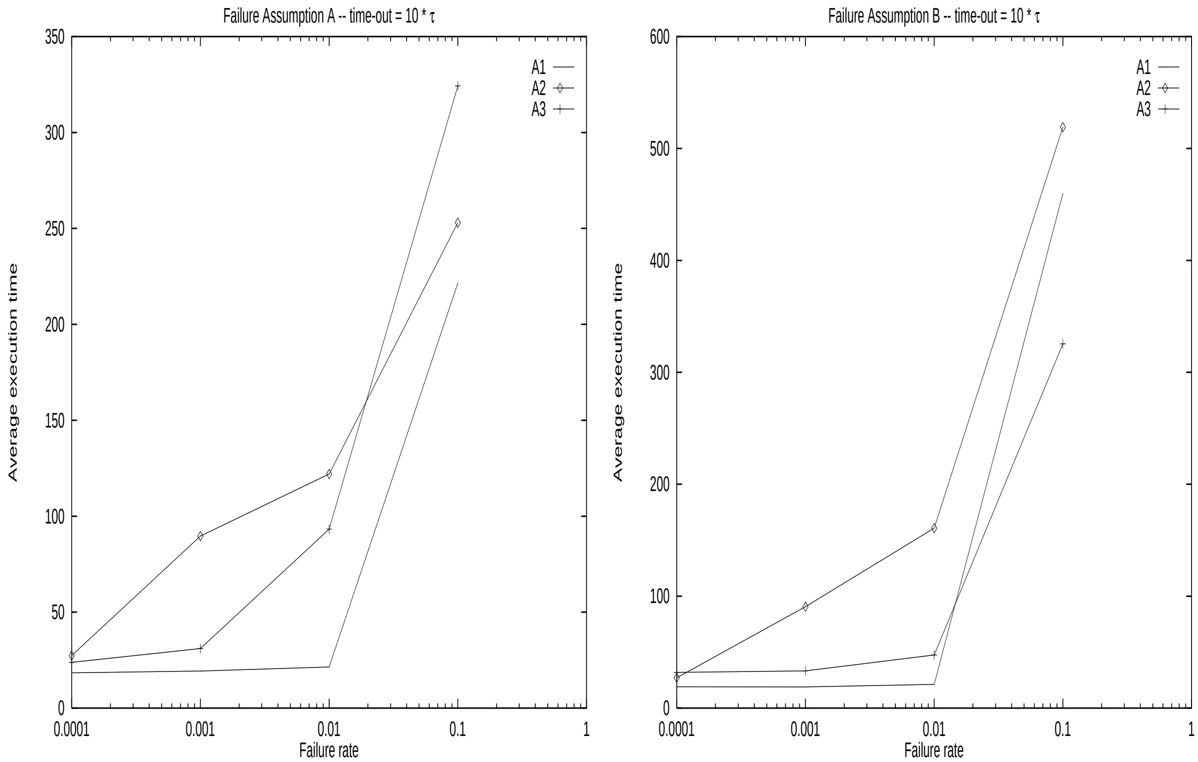


Figure 4c. Time-out of $10 * \tau$

Figure 4. Average program execution time under failure assumptions A and B

where $\omega_N = e^{j(2\Pi/N)}$ is the Nth of unity. Clearly the computation of all the components of the complex vector Y requires N^2 complex multiplications and additions. It is possible to rewrite the above equation as:

$$y(k) = y_{even}(k) + \omega^k * y_{odd}(k), \quad 0 \leq k \leq N/2 - 1$$

$$y(k + N/2) = y_{even}(k) - \omega^k * y_{odd}(k), \quad 0 \leq k \leq N/2 - 1$$

where Y_{even} is the Fourier transform of the even points of X, and Y_{odd} is the Fourier transform of the odd points of X.

The Fast Fourier Transform (FFT) [5] uses the two previous recursive equations. It computes the discrete Fourier transform of a vector using $1/2N \log_2 N$ instead of N^2 operations. We show on Figure 5 an example of a task graph for the FFT algorithm for $N = 16$ with $Nlog_2N = 64$ tasks. The connections between the tasks represent the flow of data during the computation. Initially, a source task receives the component of X whose index is the bit-reversal-permutation of the task number decremented by 1. For example, the task 2 ($2 - 1 = 1$ being coded 0001 on $log_2N = 4$ bits) will receive the component $x(8)$. Then each task performs one complex multiplication and one complex addition. If each task has an execution time t, the total execution time of the parallel FFT algorithm is $log_2N * t$ time units with at least N processors.

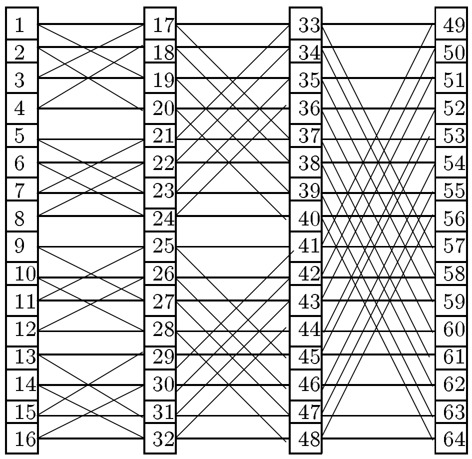


Figure 5. Task graph for the FFT computation

We have run simulations for the case where $N = 256$ generating a graph with 2048 tasks and we have used $M = 256$ processors. Each task has an execution time t corresponding to one complex multiplication and one complex addition; we have taken $3.24E - 04$ for the average duration of a complex multiplication and $1.04E - 04$ for the average duration of a complex addition (in seconds); these are realistic figures related to current workstation technology. The detection step duration w is taken equal to $6.5E - 05$ which corresponds to the average time necessary to test the state of a task. The restarting step duration C

is taken equal to $10E-05$; this is a relatively small value which implies that a processor's registers can be very rapidly loaded with the status of a task.

For all the simulations presented, we have computed confidence intervals with the probability of 95 %. The intervals we have thus obtained range from 0 % to 10 % of the average program execution times for all the simulations. They are not directly shown on the figures in order not to clutter the information which is being presented.

We easily see that when no detection algorithm is running, and when failures cannot occur, the total execution time of the graph with $N = 256$ is $(\log_2 N) * t = (\log_2 256) * (3.24E-04 + 1.04E-04) = 3.424E-03$ time units. However when there are no failures but a detection algorithm *is* running, this will generate obvious overhead.

In order to analyze the behavior of the detection algorithms, we have evaluated the FFT execution time with each detection algorithm for three values of the failure rate γ: 0.001, 0.005 and 0.01. We first consider the failure assumption A (FAA) where a task running the detection algorithm is supposed not to fail until it has completed the detection. We have taken for the maximum program duration allowed t_{max} the value 0.5 for $\gamma = 0.001$, 1 for $\gamma = 0.001$ and 1.5 for $\gamma = 0.001$.

We show on Figure 6 the average execution times for the parallel FFT algorithm for each of the failure detection and recovery techniques which we consider, as a function of the failure rate. For Algorithms 2 and 3 only the value of π which gave the shortest execution time is shown for each value of failure rate; the value $\pi = 0.25$ always provided the smallest execution time for any value of γ.

Algorithm 2 gives the smallest execution times for any value of γ, even though we observe that it results in average job execution times which are very close to Algorithm 3 for low values of γ (e.g. $\gamma = 0.001$).

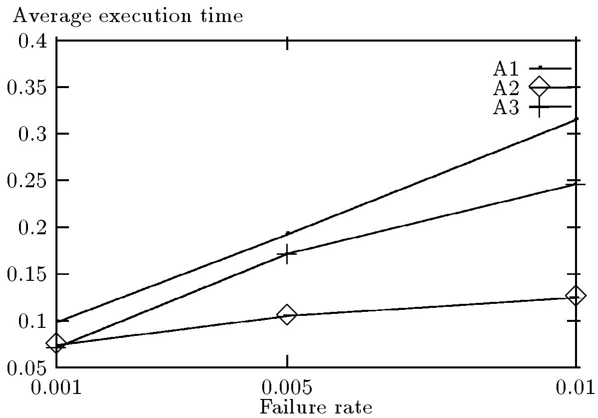


Figure 6. Comparison of the detection algorithms for FFT application

So far, we have assumed in all of the simulation results presented above that tasks will communicate "instantaneously" with other tasks, i.e. that the communication delay between tasks is negligible so that $Z = 0$. We will now explicitly address the effect of communication overhead via a serie of simulation runs carried out under the same conditions as before, with $Z = 0$ and also with $Z = 4.28E - 04$ as shown on Figure 7. We present the average execution times obtained with Algorithm 2 and $\pi = 0.25$ with and without communication overhead. We see that the additional slow-down introduced by communication overhead increases substantially with failure rate, because (as expected) as tasks are restarted, all their communication activities have to be repeated. For $\gamma = 0.01$ we observe an increase in total effective execution time of more than 30 %. Note again that all simulations produce as previously confidence intervals of less than 10 % with a confidence level which is better than 95 %.

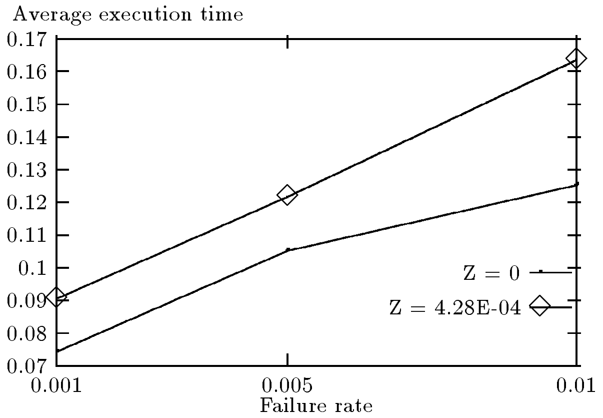


Figure 7. Execution times with and without communication overhead

A1	9.835E-02 2.07E-01			
π	0.25	0.5	0.75	1
A2	7.426E-02 9.88E-02	1.069E-01 1.398E-01	1.116E-01 1.911E-01	1.739E-01 2.506E-01
A3	7.153E-02 4.393E-01	1.034E-01 5.566E-01	3.125E-01 3.862E-01	3.632E-01 5.978E-01

Figure 8. Average execution times under FAA and FAB with $\gamma = 0.001$

Finally let us consider Failure Assumption B which allows tasks to fail as they are carrying out failure detection and recovery. On Figure 8 we provide two execution values for the average execution time for each detection algorithm with $\gamma = 0.001$; the first value was obtained under FAA and the one given below concerns FAB. As expected, in each case, the average execution time corresponding to FAB is larger than the result corresponding to FAA. However the relative difference varies both with the number of agents (as represented by the parameter π) and the time it takes to detect failures.

5 Conclusions

In this paper we have analyzed the behavior of parallel programs represented by a random task graph in a multiprocessor environment subject to failures. We have presented several algorithms which can detect failures and restart failed tasks. Our purpose was to guarantee the successful completion of parallel computations with the lowest detection and restart costs.

We have considered performance measures such as the total effective execution time in the presence of failures, and failure detection overhead. Simulations have been conducted and various parameters including the failure rate have been varied. These simulation results demonstrate the feasibility of our approach, and indicate that overhead can be maintained at a modest level.

The method we have proposed to transform a parallel program written for an ideal machine with no failures to run correctly on an unreliable machine is totally user-transparent. It is thus the work of the compiler to transform any parallel program in a resilient version able to run on a failure-prone multiprocessor architecture. This software-based dependable strategy does not require any particular hardware mechanism and allows on-line failure detection and dynamic reallocation of failed processes.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[2] B. Bhargava and S-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach. In *7th IEEE Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[3] S. Chabridon. Temps d'exécution d'un programme parallèle dans un environnement avec pannes. *Comptes-Rendus de l'Académie des Sciences de Paris*, Série II, t. 381:1315–1318, May 1994.

[4] S. Chabridon and E. Gelenbe. Dependable execution of distributed programs. In TU Delft, editor, *International Conference on Massively Parallel Processing, Applications and Development*. The Netherlands, June 1994.

[5] J.M. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[6] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th IEE Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[7] E. Gelenbe. *Multiprocessor performance*. John Wiley & Sons, 1989.

[8] E. Gelenbe. Multiprocessor speed-up and the activity set model of program behavior. In J-L. Delhaye and E. Gelenbe, editors, *High Performance Computing*, pages 121–132. Elsevier Science Publishers B.V. (North-Holland), 1989.

[9] E. Gelenbe and Z. Liu. Performance analysis approximations for parallel processing of concurrent tasks. In M. Vanneschi M. Cosnard, M.H. Barton, editor, *Parallel Processing*, pages 363–375. Elsevier Science Publishers B.V. (North-Holland), 1988.

[10] E. Gelenbe, R. Nelson, T. Philips, and A. Tantawi. Asymptotic processing time of a model of parallel computation. In *Proceedings National Computer Conference*, pages 127–138, Las Vegas - USA, 1986.

[11] Z. Kedem and K. Palem. Transformations for the automatic derivation of resilient parallel programs. In *Proceedings 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 16–25, 1992.

[12] Z. Kedem, K. Palem, M. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 306–317, 1992.

[13] Z. Kedem, K. Palem, M. Raghunathan, and A. Raghunathan. Resilient parallel computing on unreliable parallel machines. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, pages 145–172. Cambridge University Press, 1993.

[14] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22nd ACM Symposium on the Theory of Computing*, pages 138–148, 1990.

[15] K.H. Kim and H.O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–636, May 1989.

[16] R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[17] P-J. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proceedings of the 4th IEEE International Conference on Data Engineering*, pages 154–163, 1988.

[18] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[19] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, March 1983.

[20] Z. Tong, R.Y. Kain, , and W.T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed recovery. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 12–20, 1989.

[21] Y-M. Wang and W.K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 147–154, 1992.